DSL course: individual work #1

Tasks

- ☑ create data structure for regular expression syntax tree;
- ☑ function to parse expression into AST;
- ☑ create data structure for non-deterministic automata;
- ☑ create function that transforms regular expression AST into automata;
- ☑ create an application that receives the representation of regular expression and some word and checks whether the word satisfies the regexp condition.
- ☑ create data structure for deterministic automata;
- ☑ create algorithm for determinization(with removing of dead-end and unreachable states);
- ☑ create algorithm for minimization(reduction)

How to install and run

Clone/download the repository and execute `python lab1_app.py`. There is no external dependencies, only standard types and type annotations.

It works on `python3.6`, so please use the same or newer version.

Test set

There are some test sequences inside `test_cases.txt`. Just run `lab1_app.py` and input the filename.

There is `tests.py`, run it, it has other checks and examples. To understand what is going on there, open the script is required.

About the implementation

Syntax of the regular expressions

Regular expressions is a way to define some regular language(`L`).

- `a` — just any symbol, UTF-8 symbols can be used;
- `|` — logical `or` statement (priority **1**): `a|b` — `L = {a, b}`;
- `ab` — is `a+b` there `+` is concatenation (priority **2**): `ab` — `L = {ab}`;
- `a*` — `*` is Clini closure (priority **3**), it means that expression to which it is performed can occur any number of times(>= 0): `a` — `L = {e,a,aa,aaa,...}`, where `e` is empty word;
- parenthesis `(` and `)`, as usual, performs prioritization of some subexpression.

So, the following words and so on are allowed:

- `(a)(b)`
- `(ab)*`
- `a|(b*c*)*`

Escaped symbols

- `\*`

- `\(`
- `\)`
- `\|`

If the symbols are escaped, they can be used as other symbols, not a special ones. As example: regexp `\(ab\)\*` defines language of only one word `(ab)*`.

How to use the `app.py`

Create file with your regular expressions and words that should be checked in the following format:

- every line should contain 2 or 3 substrings with colon (`:`) delimiter;
- the first substring is a regular expression. Be careful, for the regular expressions all symbols matter (including spacing);
- second and third substrings are words that should be checked;
- first of them are words that must match the regexp;
- second of them are words that must not match the regexp;
- all words for checking separated by a semicolon(`;`).
- you can keep the first list of words empty, but colons should be placed.

So, the typical lines of the testing file looks like:

- `(ab)*:;ab;abab:aa;bb` — regexp + true + false words;
- `hii*:hi;hiiii` — regexp + only true words;
- `ab*::b;bb;ba` — regexp + empty + only false words.

It means, that you can't use colon and semicolon in your expressions, but it is the limitation of the current verifying application(and the scheme), not of the regexp or state machine implementations.

About regular expression parsing

After scanning (it is really simple), parsing is performed. List of tokens is recursively separated by one of the above operators with respect to their priority and parenthesis, and in this way it create AST of the expression.

Note: All the actions of parsing occur on the "first layer" of the expression, and here is what I mean.

Let's say that we have regular expression `(ab)|(c|d)`. The "first layer" of the expression is `group1|group2`, where `group1` is `(ab)`, `group2` is `(c|d)` and `|` is binary operator. Every iteration of the algorithm works only with parts of the first layer (operators inside groups are not visible by algorithm).

About translating

After parsing string to the AST, translation is performed.

There is `translate` function, that recursively performs the following steps:

- receive children for the node;
- if there are:
  - 0 child: it must be leaf with only one symbol;
  - 1 child: it must be only clini node;
  - 2 children: it can be "OR" as well as "AND";

- **here recursy goes**: it performs appropriate class method from the non-deterministic automata class: `by_value` for leaf symbols(creates automata of one symbol), `by_concatenation` or `by_decision` for machines of its children and `by_clini` for only the machine of single child.

About state machines

`automata` package implements non-deterministic(NDFA) and deterministic(DFA) finite state machines.

Finite is about number of states.

Non-determinism means that from the same state can exist more than one transition with the same "trigger" symbol.

In the other hand, determinism means the reverse.

Transitions of NDFA are stored as dictionary of dictionaries of sets (or in Python notation `Dict[int, Dict[chr, Set[int]]]`), each machine can have any number of start and final states.

Transitions of DFA are stored as `Dict[int, Dict[chr, int]]`, there are no multiple transitions of the same symbol for one state, `0` is always start state, can have multiple final states.

The implementations of machines use iteration mechanism for word verifying, not graph depth search.

Forks with transitions of the same trigger symbol are allowed in NDFA, of course. In the machine, this case is handled by multiple current states of the machine (to be exact, set of states). If fork occurs for some state, the state will be split up into multiple states, every of which will continues its "route" on its branch.

DFA works the same way, but there is only one moved state and there are no forks of the same symbol.

About algorithms of machine joining of NDFA

At first, let me define all sets:

- `Σ` is a vocabulary of symbols;
- `S` is a set of states;
- `I` is a set of initial(start) states, where `I ⊂ S`;
- `F` is a set of final states, where `F ⊂ S`;
- `T` is a set of transitions, where `T ⊂ S x Σ x S`.

So, following rules are applied to join state machines with each other(`OR` is for `|` and `AND` for `+`(concatenation)):

RegExp	Machine creation
`a ∈ Σ`: `a ∈ RegExp`	$S = \{s_0, s_1\}$, $I = \{s_0\}$ $F = \{s_1\}$, $T = \{<s_0, a, s_1>\}$
$e_1 \in RegExp$, $e_2 \in RegExp$: $(e_1\ OR\ e_2) \in RegExp$	$S = S_1 + S_2$, $I = I_1 + I_2$ $F = F_1 + F_2$, $T = T_1 + T_2$
$e_1 \in RegExp$, $e_2 \in RegExp$: $(e_1\ AND\ e_2) \in RegExp$	$S = S_1 + S_2$, $I = I_1 + I'$ $F = F_2$, $T = T_1 + T_2 + T'$ where: if $I_1 \cap F_1 \neq \varnothing$ than $I' = I_2$ else $I' = \varnothing$ and $T' = \{<s_1, a, s_2> \in S_1\ x\ \Sigma\ x\ I_2: <s_1, a, s_1'> \in T_1$ for some $s_1' \in F_1\}$
`e ∈ RegExp`: $e^* \in RegExp$	

	$S_* = S + \{N\}$, $I_* = \{N\}$ $F_* = \{N\}$, $T = T + T' + T''$ where: $T' = \{<N, a, s_2> \in \{N\} x \Sigma x S: <s_1, a, s_2> \in T$ for some $s_1 \in I\}$ $T'' = \{<s_1, a, N> \in S_* x \Sigma x \{N\}: <s_1, a, s_2> \in T'$ for some $s_2 \in F\}$

Determinization

The implementation of determinization uses powerset construction technique(see example chapter). The article above is good, but uses epsilon moves which the implementation haven't, so keep it in mind.

Minification

Minification is a process of merging of equal states.

Equal states is states that have the same paths to final states(empty path is also allowed).

The implementation uses fact, that state `s1` equals to `s2` means that if transition for `s1` by symbol `a` exists than transition for `s2` must exist and states to which they transfer must be equal, and vice versa, they are not equal if there is some not equal reachable, or there is no even transition.

For receiving equality sets, we build lower triangular matrix, where its indexes are states numbers(without dublicates and diagonal(because every state is equal to itself)).

Algorithm:

1. For every `i` and `j` state, if one of them is final and other is not(XOR), we write 1(not equal, because only final states can have empty path to some final state) in `ij` cell of the matrix.
2. Than, while we had changes in the previous iteration of this(2) step, for every `i` and `j` that have zero in `ij` cell, we check all states that are reachable by the same vocabulary symbol of the automata, and if the reachable states are not equal each other, `i` and `j` is also not equal, so we put 1(not equal) to `ij` cell.

Than in the end of the process, we will have matrix, where all equal states are marked by 0. So we can construct equality sets for all the states, and than transform previous transition graph, leaving transitions only between equality sets.